A Framework for Understanding Web Publishing Applications

Sonia Guéhis

Paris-Dauphine University,
Place du Marechal de Lattre de Tassigny, 75775 Paris, France
sonia.guehis@dauphine.fr

Abstract. We address in this paper the reverse engineering issue in the Web applications. The maintenance process of this kind of applications is often hardly performed due to the lack of documentation. In most cases, the documentation associated to the Web application does not exist or rarely complete and up-to-date. We aim to present a solution which describes the structure of Web application in order to gain their better understanding and so facilitate their maintenance. We describe a method to infer Web publishing programs, specifically defined as database-driven programs producing dynamic documents. We address a typical reverse engineering situation where the program is a "black box" that takes a database instance (the input) and produces a dynamic document (the output). Our method attempts to understand and describe the program.

Keywords: reverse engineering, Web publishing programs, dynamic documents, canonical instances

1 Introduction

1.1 Context and motivations

The production of dynamic (X)HTML documents from relational databases is probably one of the most common techniques used in Web applications development. Such documents combine *static parts* that correspond to free text and (X)HTML rendering instructions (e.g., tags), and *dynamic parts* which are retrieved at run time from a relational instance. Many specialized languages (i.e., Java/JSP, PHP, ASP) and development frameworks (Struts, .NET, PHP/MVC) make the production of dynamic Web sites a relatively easy task, and this contributes to the richness and accessibility of the Web. A downside is that publishing programs are often poorly written, and tend to be quite difficult to understand and maintain. The situation even degrades as the application evolves through maintenance and evolutions.

In the present paper we describe the main aspects of a method to address this situation. Our goal is to derive useful information on the structure and behavior of publishing programs without having to delve into the source code. The main idea is that we can infer how a program $\mathcal{P}$ accesses the underlying

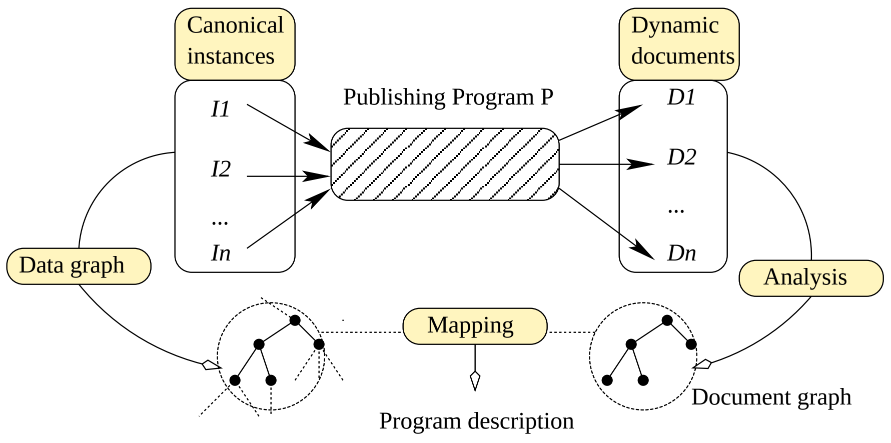

Fig. 1. Overview of the reverse engineering process

database and merges the dynamic and static parts by just examining the input and output. Moreover this re-ingeneering process needs only an access to the database schema and the right to run $\mathcal{P}$ on instances of this schema. We do *not* require an access to the actual database instance, nor do we need the code of the application. This preserves the privacy of business data, and allows to cope with situations where the source code is no longer available.

1.2 Process overview

Basically our method produces carefully chosen instances of the database, runs $\mathcal{P}$, and makes some inferences of the program behavior by analysing the dynamic document produced as output. Figure 1 illustrates the main components involved in the process. Let us briefly describe their role before entering into details.

We apply $\mathcal{P}$ to *canonical instances* of the database schema and obtain *dynamic documents*. The concept of canonical instance denotes both *complete* and *unambiguous* instances [1]. Intuitively, a canonical instance enjoys suitable properties for the analysis of the dynamic document produced by a program. For instance, given a set of values, one can determine without ambiguity the set of tuples (if such a set exists) of the instance that contain these values, as well as their dependencies. In order to model conveniently such structured sets of tuples, we view the relational instance as a *data graph* where tuples and values are nodes, and edges represent dependencies. The data graph model and canonical instances are presented in Section 2.

Next, dynamic documents are analysed in order to distinguish the static parts from the dynamic ones. This is done through an iteration of program executions that produce dynamic documents from as many canonical instances as necessary. The dynamic part is modeled as a graph of values, constructed from both the document structure and the database schema. This analysis process is described in Section 3.

Finally, we construct a *mapping* between the graph of values of a document D_i and a subgraph of the canonical instance I_i. This mapping constitutes an interpretation of the program $\mathcal{P}$ that carries out a navigation in I_i, retrieves some values and merge these values with static text to create D_i. We can then produce a description of $\mathcal{P}$ at a suitable abstraction level independent from specific details such as, for instance, the programming language. This final step is presented in Section 4.

The rest of the paper develops this brief overview, and discusses the perspective of the approach, as well as related work (Section 5). Due to space limitations, the presentation is mostly driven by examples based on a simple database that represents movies with their (unique) director and their (many) actors (Figure 2). The interested reader is referred to[1] and [2] for formal definitions and technical details on the DocQL language and our concept of complete instance. (see *http://www.lamsade.dauphine.fr/~guehis/Protos.htm*).

title	year	*id_ director*	genre
Unforgiven	1992	20	Western
Van Gogh	1990	29	Drama
Kagemusha	1980	68	Drama
Absolute Power	1997	20	Crime

Movie

id	last_name	first_name
20	Eastwood	Clint
21	Hackman	Gene
29	Pialat	Maurice
30	Dutronc	Jacques
68	Kurosawa	Akira

Artist

title	***id_ actor***	character
Unforgiven	20	William Munny
Unforgiven	21	Little Bill Dagget
Van Gogh	30	Van Gogh
Absolute Power	21	President Allen Richmond

Cast

Fig. 2. An instance of the *Movies* database

2 Modeling the input: canonical instances

As mentioned previously, we model a database instance as a labeled directed graph $\mathcal{G}_I$, and rely on a query language on this graph which constitutes a syntax-neutral (declarative) specification of a publishing program written in Java/JSP or in any other programming framework.

Data model and query language.

Our reverse-engineering process operates on a view of the relational instance where tuples are seen as internal nodes, values as leaf nodes, and edges represent either tuple-to-tuple dependencies or tuple-to-attribute dependencies. Figure 3 shows the data graph of the instance of Figure 2. We distinguish functional dependencies between nodes (e.g., between a movie node and its director node)

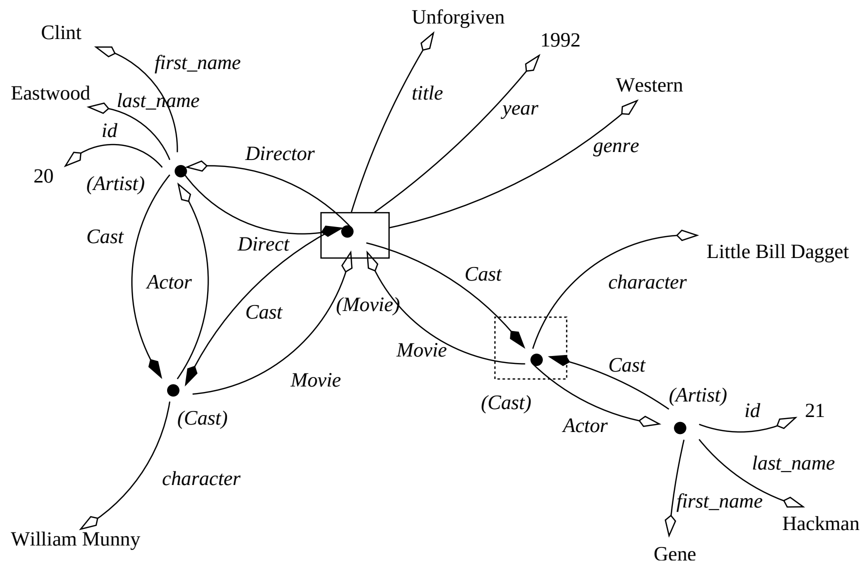

Fig. 3. A subset of the data graph of our sample instance

and multivalued dependencies (e.g., between a movie node and its actor nodes). The former are shown with white-headed arrows, the latter with black ones.

We associate to this model a query language, called DocQL, which combines *navigation* in the data graph with *instantiation* of the textual fragments that contribute to the final document. A DocQL query is essentially a tree of path expressions which denote the part of the graph that must be visited in order to retrieve the data to include in the final document. Path expressions use an XPath-like syntax. An expression p is interpreted with respect to an *initial node* N_i (unless it begins with `db` which plays the role of `/` in XPath), and delivers a set of nodes, called the *terminal nodes* of p (with respect to N_i). Each path is associated to a fragment which is instantiated for each terminal node. Path and fragments are syntactically organized in *rules* of the form `@path[condition]{fragment}`, with a path expression, a node condition and `fragment` is the fragment instantiated for each instance of `path`.

The following example shows a DocQL query over our *Movies* database. It produces a (rough) document showing the movie *Unforgiven* along with its director and actors.

```
@db.Movie[title='Unforgiven']{
  @title{}, @year{}, directed by
    @director.first_name{} @director.last_name{}
  Featuring: @Cast{
    - @artist.first_name{} @artist.last_name{}as @character{}
  }
}
```

The semantics of the language corresponds to nested loops that explore the data graph, one loop per rule. Looking at the previous example, we first search for the node *Movie* with title *Unforgiven*. Taking this node as an initial one, the value of each (unique) path `title`, `year`, etc., is evaluated. The multiple path `Cast` leads to all the nodes that represent one of the characters of *Unforgiven*. Applied to the data graph of Figure 3, one obtains the following document as result of the previous example:

```
Unforgiven, 1992, directed by Clint Eastwood, Featuring:
  - Clint Eastwood as William Munny
  - Gene Hackman as Little Bill Dagget
```

Aggregation and negation cannot be directly expressed in DocQL, but aggregated values can be obtained via the mapping that transforms the relational instance to the virtual data graph (an even simpler solution is to define SQL views with `group by` clauses, which can then be exported in the data graph). We shall discuss these limitations in Section 5.

Canonical instances.

Our method relies on the creation of specific instances satisfying two conditions: *completeness* and *non-ambiguity*. In short, the first condition ensures that the program always finds query results during its navigation in the database. The second condition is meant to allow the identification of a unique subgraph of the instance, isomorphic to the set of values found in the dynamic document.

Completeness. An instance is said *complete* if, for each one-to-many dependency $E \ ^1 \rightarrow^* \ E'$ and each tuple e instance of E, there exists an instance e' of E' associated to e. Let us take the example of the dependency *directed by* that links a director and its movies. In terms of the relational schema, there is an integrity constraint that ensures that each movie refers to a director (see Figure 2). The completeness constraint states, *in addition*, that each tuple in table *Artist* is referred to by a tuple in table *Movie*.

Therefore, in a complete instance of our schema, each artist is the director of a movie. This is by no way a realistic constraint. It is only intended to ensure that a publishing program that wishes to show a film, its actors, and for each actor, the list of films possibly directed by this actor, will produce the most complete result document. In other words a *complete* instance allows to obtain *complete* documents, and thus a *complete* view of the program output.

The instance of Figure 3 is not complete. If we remove the node squared with dashed lines (and the corresponding *Artist* subgraph), the instance becomes complete, because the only remaining artist is Clint Eastwood who turns out to be both an actor and a director. Note the cycle in the data graph that corresponds to a cyclic dependency in the graph schema.

Non-ambiguity. The *non-ambiguity* condition can be informally stated as follows: if N and N' are two nodes in the data graph, then the path that links N to N' can be uniquely determined. The instance on Figure 3 is ambiguous, even after removal of the node that corresponds to *Gene Hackman*. Indeed, if

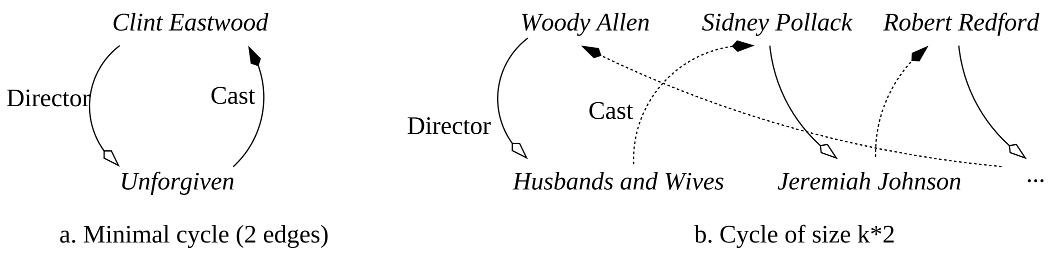

Fig. 4. Generating cycles in a canonical instance

we are given the values 'Eastwood' and 'Unforgiven' found in a dynamic document, there is an ambiguity on the meaning of the *Artist* node, which can be interpreted either as the director or an actor of the film.

Ambiguity is a consequence either of two distinct nodes sharing the same value, or of cycles in the database instance. The first problem can easily be avoided by generating distinct values. The second problem is trickier because simply removing cycles would contradict the completeness property. As mentioned above, the database needs to represent by cycles in the instance the cycles of the schema in order to obtain a solution for any path chosen by the program from any node in the graph.

A trade-off is here necessary. Note first that the cycle size in the instance is proportional to the cycle size in the schema. Figure 4.a shows a minimal cycle in our sample instance, of size 2, and Figure 4.b its generalization to a cycle of length $2 \times k$, with $k > 1$. The value of k is a parameter of the instance construction which represents the upper-bound on the number of tables joined by a single SQL query of the program (or equivalently, k is the longest path used by the program during its navigation in the data graph). k must be chosen large enough so that no ambiguity can arise when a link must be created between two values extracted from a dynamic document.

In the following we call a complete and non-ambiguous instance a *canonical instance*. An algorithm to create canonical instances is given in[1].

3 Modeling the output: dynamic documents

Assume now that we obtain a document D from the execution of the program $\mathcal{P}$ on a canonical instance I. We need to distinguish static parts from dynamic parts. From the dynamic part we will be able to construct the mapping that associates the document structure to a database subgraph. For the sake of clarity, we illustrate the mechanism with the following document D, obtained by running $\mathcal{P}$ on a canonical instance I.

```
Unforgiven, 1992, directed by Clint Eastwood, Featuring:
  <ol>
    <li> Gene Hackman, as Little Bill Dagget</li>
  </ol>
```

Some words (e.g., “Featuring”) are part of the static content, while others (e.g., “Eastwood”) come from the database. Let $\mathcal{L}(D)$ be the list of words constituting the document and $\mathcal{L}(I)$ be the list of words from the canonical instance. A first approximation is to consider that $W = \mathcal{L}(D) \cap \mathcal{L}(I)$ is the dynamic content of the document. We can then perform a full-text search in the canonical instance for each word in W, identifying the tuples which have been retrieved by the program, and the position of dynamic data in the document. Note however that the latter information may be a superset of the dynamic list, due to the presence of words in the static part which *also* appear in the instance. This would be the case for instance if the static content contains the word “little” which is found as well in our canonical instance.

This problem can be solved by using two canonical instances I and I' such that $\mathcal{L}(I) \cap \mathcal{L}(I') = \emptyset$, i.e., the instances are fully distinct (recall that we control the content of our canonical instance). Assume for instance that I' contains a description of the movie *Husbands and Wives*. The dynamic document D', resulting from the execution of the program over the canonical instance I', is:

```
Husbands and Wives, 1991, directed by Woody Allen, Featuring:
  <ol>
    <li> Sidney Pollack as Jack</li>
  </ol>
```

The list of words of the static content can be obtained as $Y = (\mathcal{L}(D) \cap \mathcal{L}(D'))$. In our example, words like “directed”, ”by”, ”Featuring” are commons words between D and D' and are parts of the static content of the publishing program.

Dynamic part of the program can be now inferred from instance I as $\mathcal{L}(D) - Y$, whereas the list for I' is $\mathcal{L}(D') - Y$. It remains to “mark” the dynamic part in one of the dynamic documents. Here is the marking for D:

```
@{Unforgiven}, @{1992}, directed by @{Clint} @{Eastwood}, Featuring:
  <ol>
    <li> @{Gene} @{Hackman}, as @{Little} @{Bill} @{Dagget}</li>
  </ol>
```

Each word w enclosed in the `@{}` tag is known to come from the instance. Moreover, since this instance is complete, we can identify the nodes in the data graph and compute the subgraph that associates these nodes, as explained below.

4 Producing the publishing program

Several algorithms for keyword-based searches in relational databases have been proposed recently, like Discover[3], DBXplorer[4] and Banks[5]. Banks seems an appropriate choice. It relies on a graph representation of the instance similar to our data graph and returns, given a set of keywords, a set of tree of tuples. The root node is called an *information node* and the tree a *connection tree*. In our case, the Banks process returns a unique tree connection, since we search

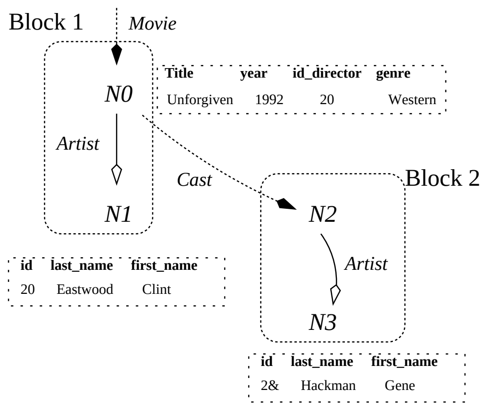

Fig. 5. Connection tree and its mapping to the associated DocQL query

keywords over a canonical instance. Banks actually represents the mapping that associates the structure of the dynamic document to the subgraph of the canonical instance. The result of this association is illustrated on Figure 5.

Four tuples have been found by the Banks algorithm, which correspond to four nodes N_0, N_1, N_2, N_3. N_0 is a *Movie* tuple, N_1 an *Artist* tuple representing the director of the movie, N_2 and N_3 are respectively the *Cast* and *Artist* representing an actor of the movie. Each edge in the graph is labeled by the table name. Recall that black-headed arrows represent one-to-many relations, whereas white-headed arrows represent one-to-one relation (i.e., a referential integrity constraint).

In order to produce the DocQL query, we group nodes in *blocks*. A block consists of a *context node* and of *satellite nodes*, containing all the nodes which have a one-to-one dependency with the context node. Figure 5 shows two blocks. The first one has for a context node $N0$, and a satellite node, N_1. The second block is composed of N_2 (context node) and N_3 (satellite node). The intuition behind the block structure is that, during its navigation in the database, the program "stops" on some node N, and produces a dynamic fragment whose dynamic part consists of all the values which are monovalued with respect to N. These values consist of (i) the attributes of the context node (e.g., the title of the movie N_0) and (ii) attributes of the nodes which have a one-to-one dependency with N (e.g., the name of the director, attribute of N_1).

A DocQL query is a tree of rules of the form @path[condition]{fragment}. One rule is constructed for each block, and the edge labels yields the structure of the query:

```
@Movie{
 @title{}, @year{}, directed by
    @artist.first_name{} @artist.last_name{}
 Featuring:
  <ol> @Cast{
      <li> @artist.first_name{} @artist.last_name{} as @character{}</li>
     } </ol>
  }
```

This ends the part of the reingeneering process which can be carried out independently from human expertise. Running the query on the canonical instances I and I' should produce documents D and D', respectively. Although this works well on this simple example, in most cases the DocQL query produced by this process will be a more or less complete approximation of the program. In the next Section we discuss how this approximation can be refined and completed by an expert user.

5 Discussion and related work

We believe that the method constitutes a good basis for a further refinement of the program description. However it is limited in many respects. A first of these limits is the expressive power of the target language. It models conjunctive SQL queries with natural joins (i.e., joins that map primary keys with foreign keys). From our experience, it is extremely rare to express non-natural joins, so we claim that this is hardly a restriction.

A second limit is a small indeterminacy in the exact limit of the blocks fragments. Looking back at the example above, there is no well-founded reason to include or exclude the tags `<ol>...</ol>` from the fragment of the block 2. This limits seems harmless, since it appears quite easy for a user to see that the `<ol>` tag occurrences are independent from the number of tuples found in the *Cast* table. An automatic recognition is probably possible in that case.

A third limit is the inability of our method to cope with constant values used in the program queries. These values can be either hardcoded in the SQL expressions in the source code, or, equivalently, provided as parameters to the program which dynamically introduce them in query expressions. The only solution is to elaborate with an expert user a list of the database fields which are subject to selection predicates in the program.

Web engineering community proposed several languages, methods and processes for the development of Web applications like WebML[6], UWE[7], Hera[8] and OO-H[9]. But, all those solutions don't address the problem of reverse engineering as we define it (namely: a known database, a known output, but an unknown program). Vaquista[10] proposes a solution to infer the presentation model of a Web page and considers only the modeling how the “static part”. A proposal of interest is WARE (Web Application Reverse Engineering) tool[11] which attempts to describe an application in terms of UML diagrams.These diagrams are helpful to understand the Web application structure. Revangie[12] is oriented toward an analysis of the user interface model, through the exploration of the Web forms of an application. proposals deal are useful for high level structures description, e.g., the graph of pages in a Web site. They can be considered as complementary of our framework which aims at giving a detailed description of a specific module.

6 Conclusion

We outlined in this paper the main principles of a reverse-engineering method devoted to Web publishing applications. The method relies on a few concepts which help to construct an interpretation of publishing programs which is both precise and language-independent. As mentioned in the discussion part, a fully automatic analysis seems impossible to achieve, but we consider the ideas presented here as a sound and promising framework for building a semi-interactive analysis tool where an expert user drives the reconstitution of a program semantic. We are currently working on an implementation of such a tool.

References

1. Guéhis, S., Gross-Amblard, D., Rigaux, P.: Publish By Example. In: 8th IEEE International Conference on Web Engineering, pp. 45–51. IEEE Computer Society Press, Los Alamitos, CA, USA (2008)
2. Guéhis, S., Rigaux, P., Waller, E.: Data-driven Publication of Relational Databases. In: 8th IEEE International Database Engineering & Applications Symposium, pp. 267–272. IEEE Computer Society Press, Delhi, India (2006)
3. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword Search in Relational Databases. In: 28th IEEE International Conference on Very Large Data Bases, pp. 670–681. Hong Kong, China (2002)
4. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A System for Keyword-Based Search over Relational Databases. In: 18th IEEE International Conference on Data Engineering, pp. 5–16. IEEE Computer Society Press, San Jose, CA (2002)
5. Bhalotia, G., Nakhe, C., Hulgeri, A., Chakrabarti, S., Sudarshan, S.: Keyword Searching and Browsing in databases using BANKS. In: 18th IEEE International Conference on Data Engineering, pp. 431–440. IEEE Computer Society Press, San Jose, CA (2002)
6. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, (2002)
7. Koch, N.: Transformation techniques in the model-driven development process of UWE. In: Workshop proceedings of the 6th IEEE International Conference on on Web Engineering, pp. 431–440. ACM, New York, USA (2006)
8. Frasincar, F., Jan.Houben, G., Vdovjak, R.: An RMM-based methodology for hypermedia presentation design. In: 5th East European Conference on Advances in Databases and Information Systems, pp. 323–337. Springer, London, UK (2001)
9. Gomez, J., Cachero, C., Pastor, O., Spain, V.: Extending a Conceptual Modelling Approach to Web Application Design. In: In 12th International Conference on Advanced Information Systems, pp. 79–93. Springer, Stockholm, Sweden (2000)
10. Vanderdonckt, J., Bouillon, L., Souchon, N.: Flexible Reverse Engineering of Web Pages with VAQUISTA. In: Working Conference on Reverse Engineering, pp. 241-248. Springer, Stuttgart, Germany (2001)
11. Di Lucca, G., Di Penta, M., Antoniol, G., Casazza, G.: An Approach for Reverse Engineering of Web-Based Application. In: Working Conference on Reverse Engineering, pp. 231-240. Springer-Verlag, Stuttgart, Germany (2001)
12. Draheim, D., Lutteroth, C., Weber, G.: A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites. In: European Conference on Software Maintenance and Reengineering, pp. 168-177. Springer, Manchester, UK (2005)